AFRL-RI-RS-TR-2011-047

DATAFLOW-BASED IMPLEMENTATION OF LAYERED SENSING APPLICATIONS

UNIVERSITY OF MARYLAND

MARCH 2011

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■**UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* March 2011	**2. REPORT TYPE** Final Technical Report	**3. DATES COVERED** *(From - To)* March 2010 – September 2010

4. TITLE AND SUBTITLE DATAFLOW-BASED IMPLEMENTATION OF LAYERED SENSING APPLICATIONS	**5a. CONTRACT NUMBER** N/A
	5b. GRANT NUMBER FA8750-10-1-0144
	5c. PROGRAM ELEMENT NUMBER T2KA
6. AUTHOR(S) Shuvra Bhattacharyya Chung-Ching Shen William Plishker Nimish Sane Hsiang-Huang Wu Ruirui Gu	**5d. PROJECT NUMBER** UN
	5e. TASK NUMBER MD
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Maryland Office of Research Administration 3112 Lee Building College Park MD 20742-5100	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/Information Directorate Rome Research Site/RITB 525 Brooks Road Rome NY 13441	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2011-047

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report describes a new dataflow-based technology and associated design tools for high-productivity design, analysis, and optimization of layered sensing software for signal processing systems. Our approach provides novel capabilities, based on the principles of task-level dataflow analysis, for exploring and optimizing interactions across application behavior; operational context; high performance embedded processing platforms, and implementation constraints. Particularly, we introduce and deliver novel software tools, called the targeted dataflow interchange format (TDIF) and Dataflow Interchange Format Markup Language (DIFML), for design and implementation of layered sensing and signal processing systems. The TDIF-CUDA (Compute Unified Device Architecture) environment is a graphics processing unit targeted software synthesis tool that provides a unique integration of dynamic dataflow modeling; retargetable actor construction; software synthesis; and instrumentation-based schedule evaluation and tuning. The DIFML package is a software package for the DIFML format, which is an Extensible Markup Language (XML)-based format for exchanging information between DIF and other tools.

15. SUBJECT TERMS
Signal processing; high performance; computer-aided design; layered sensing; dataflow.

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** STANLEY LIS
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	38	**19b. TELEPHONE NUMBER** (*Include area code*) N/A

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Summary

Below is a summary of accomplishments, listed by project task.

- API (Application Programming Interface) for GPU (Graphics Procesing Unit) Software: Accomplished. The developed APIs include those available through the Targeted Dataflow Interchange Format (TDIF) language that is used at compile time, and run-time APIs, such as the topological context (TC), execution context (EC), and first-in-first-out (FIFO) APIs.
- GPU-Targeted Software Synthesis Tool: Accomplished. In this tool, actor-specific TDIF files are parsed and APIs are generated for the corresponding actors. Actor designers can then provide the associated implementation code (in C or CUDA (Compute Unified Device Architecture)) based on the provided APIs. Furthermore, DIF (Dataflow Interchange Format) files, which are specified in the DIF language, expose the high level dataflow graph structure of the source application. DIF files are parsed by our tool, and a corresponding top-level C file is generated that implements the input dataflow graph as well as a header file that is provided for the designer to implement schedulers through a standard interface.
- Library Components, Examples, and Demonstrations: Accomplished. A fundamental image processing application centered around Gaussian filtering is demonstrated by using our TDIF-based design and synthesis approach. A collection of dataflow library components are developed to demonstrate this Gaussian filtering application.
- Instrumentation Techniques: Accomplished. To measure the performance of an actor implementation in C and CUDA, our tool is capable of invoking measurement functions provided by `gcc` and NVIDIA's SDK (Software Development Kit) for CUDA. Our tool provides measurement results by calling these functions during execution of an actor firing. In addition to demonstrating execution performance, bandwidth and memory management efficiency are also evlauated.
- DIFML (Dataflow Interchange Format Markup Language): Accomplished. DIFML is designed as an XML (Extensible Markup Language)-based format for exchanging information between the DIF language and other tools and languages, and more generally, between arbitrary pairs of dataflow environments. The associated utility in the package is designed to support bi-directional transformations between DIF files and DIFML files.

2. Introduction

Signal processing applications for layered sensing can often be described in terms of signal processing block diagrams. In early design stages, system blocks are treated as "black boxes," and designers focus on defining application specifications and features at a high level of abstraction.

After a target platform is chosen, system blocks are manually transcoded, and the resulting implementations are tuned to match the platform. Such a design process, from an initial application description to a final implementation, often consists of several complex design steps that are linked by different design languages and tools, and ad-hoc transcoding processes. While targeting heterogeneous, high performance design platforms, such a process tends to be more error-prone and time-consuming due to the need for efficient coordination across different processor types. Therefore, a cross-platform design environment is needed that provides capabilities for the designer to experiment with key design phases — ranging from early design exploration to final implementation tuning — on different platforms.

Model-based design methods based on *dataflow* models of computation have become increasingly popular to provide formal semantics for such block diagrams because of their natural correspondence to signal flow graphs and system level DSP (Digital Signal Processing) flows. Consequently, dataflow graphs are widely used to model applications in many signal processing domains (e.g., see [1]).

In dataflow models of computation, signal processing applications are modeled as directed graphs, where vertices (*actors*) represent computational modules for executing (or *firing*) functional tasks, and edges represent first-in-first-out (FIFO) channels for storing data values (*tokens*), and imposing data dependencies between actors. Whenever an actor fires, it produces and consumes tokens from its input and output edges, respectively.

Nowadays, the most popular way of generating code for high performance GPUs is through using low level specialized languages or APIs. When graphics cards were used only for graphics, programmers used OpenGL (Open Graphics Library) or shader languages like Cg (C for Graphics). But as more general purpose programs became supported, general purpose languages like CUDA or Close to Metal (now in Stream Computing SDK) have thrived for NVIDIA and AMD (ATI), respectively. These languages are C variants giving programmers a familiar front end, while restricting some C features and introducing GPU specific constructs. These allow designers the best opportunity for maximum performance as little of the architecture is abstracted. But they do create a time consuming design process due to difficult design decisions. The resulting code has limited portability and must be re-tuned for different or newer architectures. OpenCL (Open Computing Language) provides a more portable (but not much higher level) way of describing a multicore application, but at the expense of performance.

Developing applications for GPUs from higher level descriptions has been an important point of research in recent years. Streaming languages like Brook and StreamIT target GPUs using high level streaming constructs. Streaming languages leverage the parallelism within a stream and the flexibility to consume and produce an arbitrary number of tokens to maximize memory bandwidth and balance compute and input/output. When the application matches the streaming model, these approaches can work well, but they often suffer from being too restrictive, while not being able to tap into compiler advances from the low level program approaches. In order to tap more into the performance potential, recently researchers at NC State have worked to raise the abstraction level of CUDA by enabling designers to not specify certain design parameters that tie a kernel to a specific GPU. This "Naive CUDA" can then be more portable and reduce the burden on designers. However applications must still be structured in a CUDA specific way.

The dataflow based approach used in this project is unique by leveraging the power of dataflow models, but still allowing low level customizations. It provides a more flexible framework without compromising on the types of optimizations possible by offering a breadth of formal models for the application designer to choose from. This application description is then tied as closely as possible to the application domain, not the target, making it highly portable while still structured enough to be optimized for. Furthermore, individual actors can still be tuned using low level techniques. These additional models will also open the high level application descriptions to a wider gamut of application domains. This work lays the foundation for such an approach.

The Dataflow Interchange Format (DIF) framework provides a standard approach for specifying mixed-grain dataflow-based semantics for signal processing system design [2]. The DIF Language (TDL), which is part of the DIF framework, provides a unified textual language for expressing different kinds of dataflow semantics, including graph topologies, hierarchical design structure, dataflow-related design properties, and actor-specific information. TDL is therefore suitable for both programming and interchange (transfer of dataflow graphs across design tools). By using TDL, signal processing systems can be represented as dataflow graphs at a high level of abstraction.

The DIF package (TDP) is a software tool that accompanies TDL, and provides a variety of intermediate representations, analysis techniques, and graph transformations that are useful for working with dataflow graphs. With the support of module libraries for the actors referenced in a dataflow graph, an efficient software implementation for the graph can be synthesized automatically using the DIF-to-C tool [2]. Although DIF-to-C supports only static dataflow applications — in particular, those that are based on synchronous dataflow (SDF) semantics [3] — the tool is capable of exploring a wide range of useful implementation trade-offs that are exposed effectively through DIF-based dataflow representations.

In this project, we develop new dataflow-based technology and associated design tool, called the *targeted dataflow interchange format* (*TDIF*), for high-productivity, high-confidence design and optimization of layered sensing software. TDIF-CUDA is a specialized variant of TDIF that is geared towards GPU-based implementation. TDIF-CUDA provides a new GPU-targeted software synthesis environment for generating software implementations that can be compiled onto CUDA-enabled GPU platforms.

TDIF extends the capabilities of DIF with dynamic dataflow software synthesis, cross-platform actor design support, and dataflow-integrated features for instrumenting and tuning implementations. More broadly, TDIF provides novel capabilities, based on the principles of task-level dataflow analysis, for exploring and optimizing interactions across application behavior; operational context; heterogeneous platforms, including high performance embedded processing architectures; and implementation constraints. Our approach provides a formal basis for systematic integration of embedded signal processing software on high performance platforms for layered sensing and information processing.

We demonstrate the TDIF environment using a Gaussian filtering application for image processing. We also describe how the designer can use the application programming interface generated using the TDIF environment to design and develop libraries of actors associated with application dataflow graphs as well as schedulers for executing the applications.

It is worth noting that there is no inherent restriction on the complexity of the application specifications that can be handled by our tool. The allowable complexity of an implementation that is derived from our tools is limited by the characteristics of the targeted GPU platform. Furthermore, our tool can, in general, handle dataflow graphs that have both static and dynamic application behavior. This is useful, for example, to support environmental awareness functionality, where dynamic dataflow support is important.

3. Methods, Assumptions, and Procedures

3.1. Application Programming Interface

3.1.1. The TDIF Language

In order to provide high-level specifications for writing dataflow actors that can be retargeted across different platforms, we have developed a first version of a language called the *TDIF language*. In our previous work, we developed preliminary underpinnings for TDIF and associated demonstrations for application specific integrated circuit implementation based on the Verilog hardware description language (HDL). TDIF is based on the *core functional dataflow* (CFDF) [4] model of computation, which provides a generalized modeling framework that is suitable for efficient representation, analysis, and scheduling of signal processing systems.

The TDIF language gives a high level specification format for writing dataflow actors that can be efficiently and reliably retargeted across different platforms. The TDIF language is a light-weight language that consists of five keywords: `module`, `input`, `output`, `param`, and `mode`.

A given actor specification should contain (at the beginning) a single `module` statement; each of the other kinds of statements can be repeated as many times as needed for the given type of structure being declared (e.g., two `input` statements and one `output` statement for a two-input, single-output actor).

The keyword `module` defines an actor with name and type that specifies the targeted language used to implement this actor. The syntax of `module` is:

`module` <type> <actor name>

For example,

```
module CUDA inner_product
```

defines an actor module named inner_product, and the targeted language for implementing this actor is CUDA.

The keyword `input` defines input ports of an actor with names and token types with respect to the associated FIFOs. The syntax of `input` is:

`input` <name of input port> <token type>

For example,

```
input input1 float
input input2 float
```

defines two input ports: `input1` and `input2` of an actor. Both types of tokens that are stored at the associated FIFOs linked to `input1` and `input2` ports are floats.

Similarly, the keyword `output` defines output ports of an actor with names and token types with respect to the associated FIFOs. The syntax of `output` is:

`output` <name of output port> <token type>

For example,

```
output output1 float
```

defines one output port: `output1` of an actor. The type of tokens that are stored at any FIFO linked to `output1` port is `float`.

The keyword `param` defines parameters of an actor with names and the associated parameter types. The syntax of `param` is:

`param` <parameter name> <parameter type>

For example,

```
param X int
param Y float
param Z char
```

defines three parameters: `X`, `Y`, and `Z` of an actor. Types of these parameters are integer, floating point, and character, respectively.

The keyword `mode` defines modes of an actor with names based on CFDF semantics. The syntax of `mode` is:

`mode` <mode name>

For example,

```
mode INACTIVE
mode PROCESS
```

defines two modes: `INACTIVE` and `PROCESS` of a CFDF actor.

In the TDIF environment, an actor invocation has an operational context, which is encapsulated by its *execution context* (*EC*), and a *topological context* (*TC*) or *dataflow context*, which is encapsulated by a list or array of incident ports.

3.1.2. Topological Context

An actor's topological context (TC) defines lists of input and output ports, along with their associated FIFO buffers, for an actor. Functions that associate the given FIFO to the ports of a TC are implemented as well as the functions that perform read and write operations to a TC at the given ports.

We provide interfaces for using C-based TCs of actors. Implementation of these interfaces has been integrated as part of the run-time library in the TDIF environment. Descriptions of these interfaces are available in the delivered `tdifc_tc.h`.

3.1.3. Execution Context

An execution context (EC) includes a special state variable, which is common to all actors, and keeps track of the current functional mode associated with the context; a special parameter, also common to all actors, that implements the vectorization (block processing) degree of the context;

function pointers for actor invocation, and data rate computation; the set of parameters for the context; and the set of state variables for the context.

In an EC, two functions characterize execution of an actor, the invoke function, and data rate computation function. Actor designers are required to follow the associated application programming interfaces and provide specific implementations for both functions. This provides a structured methodology for developing actors that can be formally integrated with the overall DIF framework.

Checking for fireability (whether or not a dataflow actor has sufficient input data to perform a quantum of computation) can be implemented automatically from knowledge of the current actor mode, FIFO populations of the input ports, and date rate computation functions. Therefore, fireability checking is not implemented by the actor designer nor stored separately for individual actors. This architecture treats the invoke and data rate computation functions as additional "special parameters" that can also conceivably be changed through dynamic parameter management.

We provide interfaces for using C-based ECs of actors. Implementation of these interfaces has been integrated as part of the run-time library in the TDIF environment. Descriptions of these interfaces are available in the delivered `tdifc_ec.h`.

3.1.4. FIFO Context

A run-time FIFO library for communication between CUDA-based, GPU-targeted dataflow actors has also been developed. This library includes both APIs and implementation code. Each data item in the FIFO is referred to as a "token". For a given FIFO instance, there is a fixed token size (number of bytes per token). Tokens can have arbitrary data types — e.g., they can be integers, floating point values (`float` or `double`), characters, or pointers (to any kind of data). This organization allows for flexibility in storing different kinds of data values, and efficiency in storing the data values directly (without being encapsulated in any sort of higher-level "token" object).

We provide interfaces for using C-based dataflow FIFOs. Implementation of these interfaces has been integrated as part of the run-time library in the TDIF environment. Descriptions of these interfaces are available in the delivered `tdifc_fifo.h`.

3.1.5. Design Template for Dataflow Actors

In the TDIF environment, a well-structured design template is provided as an API for writing dataflow actors based on the interfaces and run-time libraries for ECs and TCs. These templates will be generated automatically after an associated tdif file is compiled. Here, `<actor name>` indicates the placeholder of a name with respect to an actor.

The invoke function executes an actor instance of a library module with a given execution context and a given topological context. The API of the invoke function is:

```
static void tdifcuda_lib_<actor name>_invoke(tdifc_tc_pointer tc,
        tdifc_ec_pointer ec);
```

The production rate function enables querying of production rates with respect to the associated modes in an actor. It returns the production rate for an actor at the given output port, and for the given execution context. The API of the production rate function is:

```
static int tdifcuda_lib_<actor name>_prod_rate(int output_index,
        tdifc_ec_pointer ec);
```

Similarly, the consumption rate function enables querying of consumption rates with respect to the associated modes in an actor. It returns the consumption rate for an actor at the given output port, and for the given execution context. The API of the consumption rate function is:

```
static int tdifcuda_lib_<actor name>_cons_rate(int input_index,
        tdifc_ec_pointer ec);
```

Th initial design function initializes a designer-generated module (actor template). The API of the initial design function is:

```
static void tdifcuda_lib_<actor name>_module_init_des(
          void *args);
```

The free design function finalizes a designer-generated module (actor template). The API of the free design function is:

```
static void tdifcuda_lib_gfilter_module_free_des(void);
```

It is worth noting that the run-time libraries for ECs, TCs, and FIFOs are implemented in C. Therefore, as a naming convention, all file names in these run-time libraries are prefixed with `tdifc`. For user-specified actors and schedulers, we use `tdifcuda` as a file name prefix because they can be implemented in either C or CUDA, and CUDA can be employed as a wrapper for C. At this level, for C-based actors, we are not dealing with kernel acceleration but rather with overall schedule coordination.

3.2. GPU-targeted Synthesis Tool

The TDIF environment currently supports C- and GPU-based implementations (i.e., for CPU and GPU platforms). The GPU-based capabilities of TDIF are currently oriented towards NVIDIA GPUs, based on the CUDA programming framework [5]. Since CUDA is a C-like programming language (CUDA can be viewed a variant of C with NVIDIA extensions and certain restrictions), a C- or CUDA-based actor can be implemented as an abstract data type (ADT) to enable efficient and convenient reuse of the actor across arbitrary applications. In typical C implementations, ADT components include header files to represent definitions that are exported to application developers and implementation files that contain implementation-specific definitions.

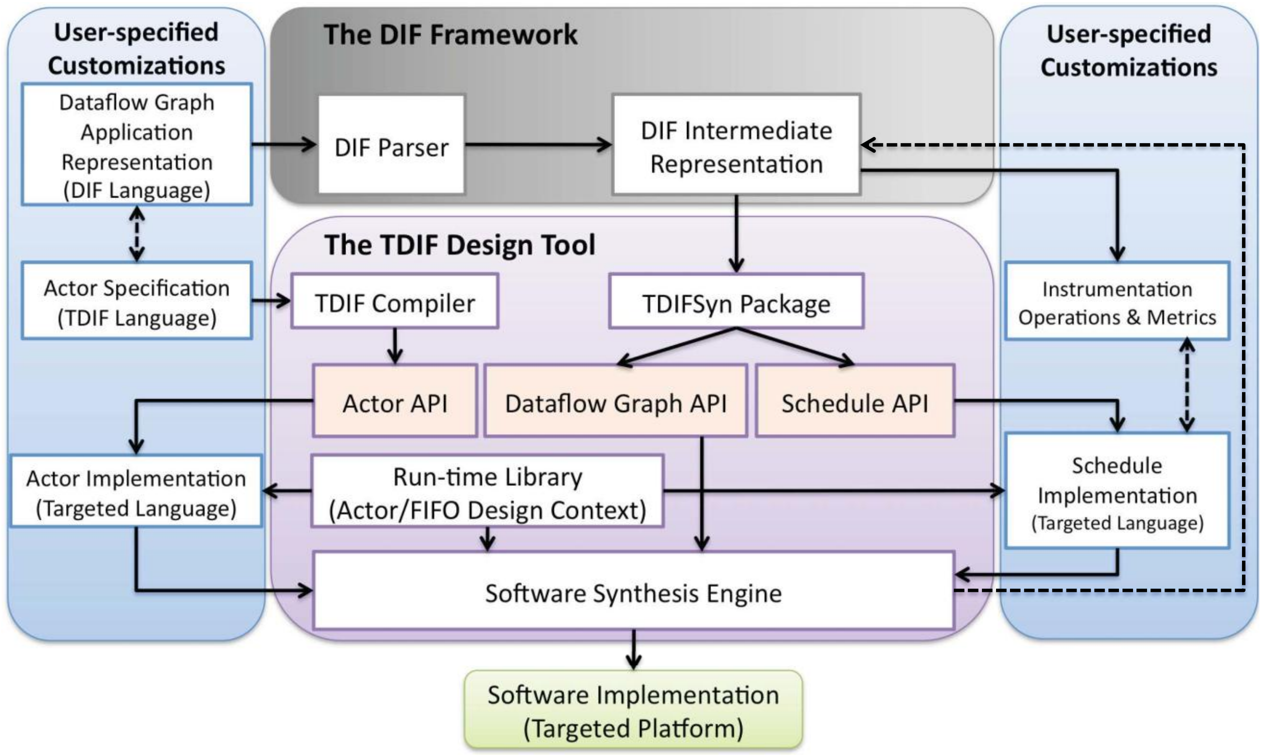

User-specified Customizations
The DIF Framework
User-specified Customizations
Dataflow Graph Application Representation (DIF Language)
DIF Parser
DIF Intermediate Representation
The TDIF Design Tool
Actor Specification (TDIF Language)
TDIF Compiler
TDIFSyn Package
Instrumentation Operations & Metrics
Actor API
Dataflow Graph API
Schedule API
Actor Implementation (Targeted Language)
Run-time Library (Actor/FIFO Design Context)
Schedule Implementation (Targeted Language)
Software Synthesis Engine
Software Implementation (Targeted Platform)

Figure 1: TDIF-based Design Flow

An illustration of the TDIF environment and associated design flow is shown in **Figure 1**. By following this methodology, the designer can focus on design implementation and optimization for dataflow actors and experiment with alternative task scheduling strategies and instrumentation techniques for the targeted platforms based on programming interfaces that are automatically generated from the TDIF tool. These automatically-generated interfaces provide well-defined, structured design templates for the designer to follow in order to generate dataflow-based actors that are formally integrated into the overall synthesis tool. In **Figure 1**, the dashed line indicates design considerations that need to be taken into account jointly to achieve maximum benefit from TDIF-based system design.

The TDIF environment is based on four software packages — the *TDIF compiler*, *TDIFSyn (TDIF Synthesis)* software synthesis package, *TDIF run-time library*, and *Software Synthesis Engine*. The interactions among these packages are illustrated in **Figure 1**.

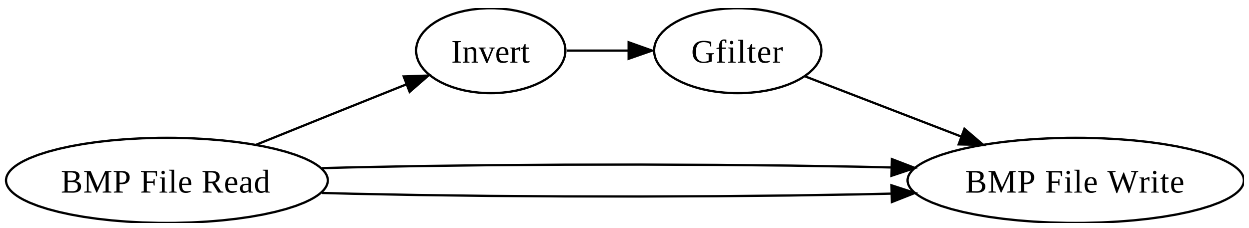


Figure 2: Application Graph for Image Processing Using Gaussian Filtering

```
module CUDA gfilter

output output1 float

input input1 float

param tileX int
param tileY int
param filter size int
param grid size int
param block size int

mode init
mode filter
```

Figure 3: TDIF Specification for the `Gfilter` Actor

The TDIF compiler, which is developed based on the Bison compiler construction framework [6], parses the TDIF specification of an actor and generates corresponding application programming interfaces (APIs) for CFDF-based, dataflow implementation of the actor in the targeted language. For C and CUDA, these APIs are generated in the form of header files for the actor programmer to base his or her implementations on. The APIs provide standard prototypes for interface functions, including the `invoke` function, which implements the functionality of the actor, and two data rate functions that return the production rate and consumption rate, respectively, associated with a given port and a given mode. The generated API features also include relevant constant definitions associated with the dataflow actor, including the numbers of input ports, output ports, modes, and parameters.

In the software deliverables for the project, the command that is used to perform CUDA-oriented compilation for the TDIF language is

`tdifcuda` <input tdif file>

The compiler output includes auto-generated header files for a GPU-targeted actor.

The TDIFSyn package is a Java package that takes a DIF intermediate representation as input from the DIF framework (e.g., a representation that has been constructed from a TDL file), and generates a top-level C language implementation file and associated API for schedulers. Here, by scheduling, we mean the assignment of dataflow actors to processors and the execution ordering of actors that share the same processor. Extensive prior work exists on scheduling dataflow graphs for various purposes (e.g., see [1]). However, systematic techniques are lacking for transferring the results of scheduling techniques into practical implementations. TDIFSyn helps to bridge this gap by providing target-language-specific APIs through which scheduling results can interact with the dataflow graph and its individual components.

The automatically generated top-level C file initializes the *operational contexts* of actors and FIFOs (communication channels between actors), which have been described in Section 3.1; configures actor parameters; lays out the graph topology by instantiating connections between actor ports and their incident FIFOs; and calls a user-defined scheduler that is implemented based on the generated scheduling API.

In the software deliverables for the project, the command that is used to compile the DIF language and generate a top-level C file is

`tdifsyn` <input dif file> <output C file>

The generated C code implements the input dataflow graph and a header file for designers to implement schedulers.

3.3. Library Components and Application Example

In Section 3.2, we have described our GPU-targeted synthesis tool, while in Section 3.1, we have provided the details of associated APIs. In this section, we focus on how to use these interfaces to develop libraries of actors, which can be systematically integrated with the overall TDIF-based synthesis tool.

3.3.1. Application: Gaussian Filtering for Image Processing

We use a simple image processing application centered around Gaussian filtering to demonstrate our TDIF-based design and synthesis approach. **Figure 2** shows a graphical representation for this application. A bitmap (BMP) image file is read by the source `BMP_File_Read` actor. This actor converts the input image into a number of tiles that are smaller in size compared to the original image. During its firing, the actor writes one of the tiles to the output buffer. The actors `Invert` and `Gfilter`, which invert the input bitmap image and apply a Gaussian filter, respectively, operate on input tokens that encapsulate tiles. The `BMP_File_Write` actor creates an output bitmap image of a size equal to that of the original image using the processed tiles.

Two-dimensional Gaussian filtering is a common kernel in image processing used for smoothing, denoising, etc. Filtering the image using a Gaussian filter involves a two-dimensional convolution operation between the image and the filter. Such operations on image pixels are attractive candidates for implementation on GPUs.

```
module CUDA bmp_file_read

output tile out float
output newrow out int
output bmpinfo out bmp_file_info

param file FILE
param tileX int
param tileY int
param halo int

mode init
mode read
mode idle
```

Figure 4: TDIF Specification for the `BMP_File_Read` Actor

A TDIF specification for the `Gfilter` actor is shown in **Figure 3**. The actor specification is parameterized to allow high level experimentation. This is reflected in the TDIF specification shown **Figure 3**, where parameters are identified using the keyword `param`. Using the parameterization features of the actor, the application designer can specify the number of tiles into which the image should be divided along with the size of the filter. At the same time, parameters specific to GPU implementation such as grid and block sizes can also be specified at a high level.

To apply the Gaussian filtering actor to a tile, input data is padded with a limited neighborhood around it (called a *halo*) depending upon the `filter_size`. Therefore, tiles produced by `BMP_File_Read` overlap. The *halo* is discarded after Gaussian filtering. The main processing pipeline in the graph is single-rate in terms of tiles and can be statically scheduled, but after initialization and end of file behavior is modeled, there is conditional dataflow behavior in the application graph.

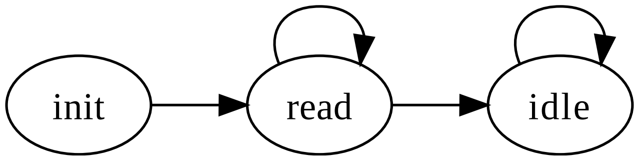
3.3.2. Actor Design

We demonstrate actor code development using the `BMP_File_Read` actor. **Figure 4** shows a TDIF specification for this actor. This is a source actor, and hence, does not have any inputs associated with it. We model this actor using the core functional dataflow (CFDF) model [4]. The actor has three different modes `init`, `read`, and `idle`. The functionality associated with each of these modes is described below. Note that the dataflow behavior, although fixed for a given CFDF mode, can in general vary across different CFDF modes of the same actor.

As mentioned in Section 3.1, in the TDIF environment, CUDA can just serve as a wrapper for C. Therefore, in a TDIF specification, we also can specify CUDA as the targeted language for C-based actors, and at this level, we are not dealing with kernel acceleration but rather with overall schedule coordination.

Mode	Production rate (number of tokens)		
	tile_out	newrow_out	bmpinfo_out
init	0	0	1
read	1	1	0
idle	0	0	0

(a) Dataflow Behavior in CFDF Modes



(b) CFDF Mode Transition

Figure 5: CFDF Modeling for the `BMP_File_Read` Actor

When fired in the `init` mode, the BMP File Read actor sets the parameters specified in its TDIF specification. It also reads the input bmp file specified by parameter `file`. This file contains two components — information about the bmp file and the actual data representing the image. In this mode, the actor outputs the bmp file information on its output `bmpinfo_out`. It also allocates sufficient memory internally to store a tile of size *(tileX+2×halo)×(tileY+2×halo)*. It does not output anything on the other two outputs. It always returns the `read` mode as the next mode in which the actor must be fired.

The BMP File Read actor when fired in the `read` mode, creates a tile of size *tileX×tileY* from the original input image. It then pads this tile by *halo* number of rows and columns around its edges. The actual values at these pixels are used during the padding. The tiles residing on the outer borders of the image for which no data values are available for padding are zero-padded. The actor outputs such padded tiles onto the output `tile_out`.

The image is processed row-wise starting from the tile containing the pixel located at index *(0,0)* (the top-left corner of the image) and proceeding along the first row of the image. When the last tile along the first row of the image is output, the tile with its top-left index coinciding with pixel *(tileX,0)* in the original image is formed and output. A similar procedure is repeated until the entire image is processed. The beginning of a new row of tiles is indicated by outputting 1 onto the output `newrow_out`. A token with value 0 is output on this edge at all other times. This mode always returns back to the same mode until the entire image has been processed, after which it returns the `idle` mode as the next mode of firing.

The actor, when fired in the `idle` mode, performs no functional computation, and remains in this mode unless forced by the scheduler to fire in a different mode. **Figure 5** (a) shows the dataflow behavior of the actor `BMP_File_Read` in all of its CFDF modes, while **Figure 5** (b) shows the possible mode transition behavior for the actor. The designer, who wants to develop the code for a new actor in a library of actors, has to translate the dataflow and functional behavior of an actor to appropriate methods available in the actor's API listed in Section 3.1.5.

We remind the reader that each CFDF actor has fixed consumption and production rates for a given mode. The methods

```
static int tdifcuda_lib_<actor name>_cons_rate(int input_index,
         tdifc_ec_pointer ec);
```

and

```
static int tdifcuda_lib_<actor name>_prod_rate(int output_index,
         tdifc_ec_pointer ec);
```

return the number of tokens consumed and produced by the actor in a particular mode for the specified input and output, respectively. The code for the method `tdifcuda_lib_bmp_file_read_prod_rate` of the actor `BMP_File_Read`, for example, is as shown below

```
static int tdifcuda_lib_bmp_file_read_prod_rate(int output_index,
        tdifc_ec_pointer ec) {
    int prod = 0;
    if (tdifc_ec_get_mode(ec) ==
            TDIFC_LIB_BMP_FILE_READ_M_IDX_IDLE) {
    } else if (tdifc_ec_get_mode(ec) ==
            TDIFC_LIB_BMP_FILE_READ_M_IDX_INIT) {
        if (output_index ==
            TDIFC_LIB_BMP_FILE_READ_O_IDX_BMPINFO_OUT) {
            prod = 1;
        }
    } else if (tdifc_ec_get_mode(ec) ==
            TDIFC_LIB_BMP_FILE_READ_M_IDX_READ) {
        if (output_index ==
                TDIFC_LIB_BMP_FILE_READ_O_IDX_TILE_OUT ||
                output_index ==
                TDIFC_LIB_BMP_FILE_READ_O_IDX_NEWROW_OUT ) {
            prod = 1;
        }
    } else {
        tdifcuda_lib_bmp_file_read_invoke_error(
                "Invalid actor mode");
    }

    return prod;
}
```

The functionality of an actor in each of its CFDF modes is coded into the invoke method:

```
static void tdifcuda_lib_<actor name>_invoke(tdifc_tc_pointer tc,
        tdifc_ec_pointer ec);
```

The designer must ensure that the code for this method conforms to the dataflow behavior of the actor as specified by the associated production and consumption rate methods. The following code implements the `invoke` method of the `BMP_File_Read` actor.

```
static void tdifcuda_lib_bmp_file_read_invoke(
        tdifc_tc_pointer tc, tdifc_ec_pointer ec) {
    int mode = TDIFC_MODE_NULL;
    int next_mode = TDIFC_MODE_NULL;

    static int tileX = 0;
    static int tileY = 0;
    static int halo = 0;
    FILE *file = NULL;

    int x = 0;
    int y = 0;
    int newrow = 0;

    static float *newtile = NULL;
    static bmp_file_info bmpinfo;

    static unsigned char *data = NULL;
    static int imgDimX = 0;
    static int imgDimY = 0;
    static int tileIndexX = 0;
    static int tileIndexY = 0;

    /* Perform the appropriate computation based on the current
       mode. */
    mode = tdifc_ec_get_mode(ec);
    if (mode == TDIFC_LIB_BMP_FILE_READ_M_IDX_INIT) {
        /* Read the header information from the bmp file */
        file = ((FILE *)tdifc_ec_get_param(ec,
                TDIFC_LIB_BMP_FILE_READ_P_IDX_FILE));
        fread(&(bmpinfo.bmptype), sizeof(unsigned short), 1,
            file);
        fread(&(bmpinfo.bmpheader), sizeof(bmp_file_header), 1,
            file);
        fread(bmpinfo.pallet, 4,
            bmpinfo.bmpheader.header.num_colors, file);

        data = malloc(bmpinfo.bmpheader.header.width *
                bmpinfo.bmpheader.header.height * sizeof(byte));

        /* Read the data from the bmp file */
        fread(data, sizeof(byte),
                bmpinfo.bmpheader.header.width *
                bmpinfo.bmpheader.header.height, file);

        /* Get parameter values. */
        tileX = *((int *)tdifc_ec_get_param(ec,
                TDIFC_LIB_BMP_FILE_READ_P_IDX_TILEX));
        tileY = *((int *)tdifc_ec_get_param(ec,
                TDIFC_LIB_BMP_FILE_READ_P_IDX_TILEY));
        halo = *((int *)tdifc_ec_get_param(ec,
                TDIFC_LIB_BMP_FILE_READ_P_IDX_HALO));

        /* Dimensions in tiles - rounded down */
        imgDimX = bmpinfo.bmpheader.header.width / tileX;

        /* Dimensions in tiles - rounded down */
        imgDimY = bmpinfo.bmpheader.header.height / tileY;

        /* Form a new tile */
        newtile = malloc(sizeof(float) * (tileX + halo * 2) *
                (tileY + halo * 2));
        /* Write the output. */
        tdifc_tc_write(tc,
                TDIFC_LIB_BMP_FILE_READ_O_IDX_BMPINFO_OUT,
                &bmpinfo);
        next_mode = TDIFC_LIB_BMP_FILE_READ_M_IDX_READ;
    } else if (mode == TDIFC_LIB_BMP_FILE_READ_M_IDX_READ) {
        /* Form a new tile */
        newtile = malloc(sizeof(float) * (tileX + halo * 2) *
                (tileY + halo * 2));

        /* Assume the last index points to the right part of m
           emory */
        for (y = 0; y < tileY + 2 * halo; y++) {
            for (x = 0; x < tileX + 2 * halo; x++) {
                float val = 0;
                if (!((tileIndexX == 0) && (x < halo)) &&
                        !((tileIndexY == 0) && (y < halo)) &&
                        !((tileIndexX == imgDimX - 1) &&
                        (x + 1 > halo + tileX)) &&
                        !((tileIndexY == imgDimY - 1) &&
                        (y + 1 > halo + tileY))) {
                    val = data[(bmpinfo.bmpheader.header.width) *
                            (tileIndexY * tileY + y - halo) +
                            (tileIndexX * tileX + x - halo)];
                }
                newtile[(tileX + 2 * halo) * y + x] = val;
            }
        }
        /* Check for new row */
        if (tileIndexX == imgDimX - 1) {
            newrow = 1;
        } else {
            newrow = 0;
        }

        /* Write to output buffers */
        tdifc_tc_write(tc,
                TDIFC_LIB_BMP_FILE_READ_O_IDX_TILE_OUT,
                &newtile);
        tdifc_tc_write(tc,
                TDIFC_LIB_BMP_FILE_READ_O_IDX_NEWROW_OUT,
                &newrow);

        /* Increment tile indices */
        tileIndexX++;

        if (tileIndexX >= imgDimX) {
            tileIndexX = 0;
            tileIndexY++;
        }

        /* Determine next mode */
        if (tileIndexY >= imgDimY) {
            next_mode = TDIFC_LIB_BMP_FILE_READ_M_IDX_IDLE;
        } else {
            next_mode = TDIFC_LIB_BMP_FILE_READ_M_IDX_READ;
        }
    } else if (mode == TDIFC_LIB_BMP_FILE_READ_M_IDX_IDLE) {
        next_mode = TDIFC_LIB_BMP_FILE_READ_M_IDX_IDLE;
    } else {
        tdifcuda_lib_bmp_file_read_invoke_error(
                "Invalid actor mode");
    }

    /* Set the next mode. */
    tdifc_ec_set_mode(ec, next_mode);
}
```

This method provides functionality for the core computational component of each CFDF actor mode. The code for each of the CFDF modes consists of — (1) consuming the required number of tokens (unless the actor is a source actor), from the input buffers; (2) processing any consumed tokens; and (3) producing the required number of output tokens (unless the actor is a sink actor) onto the output buffers; and (4) returning the next mode in which the actor should be fired. The last component of the code effectively translates the mode transition behavior into the actual actor design.

We emphasize that code for certain types of actors may not have all of the first three components. For example, a source actor does not have any input buffers, and correspondingly, does not consume any tokens, while a sink actor does not have any outputs, and hence, produces no tokens. As another example of an `invoke` method, we provide below CUDA code for the `invoke` method of the `Invert` actor.

```
static void tdifcuda_lib_invert_invoke(tdifc_tc_pointer tc,
         tdifc_ec_pointer ec) {
    int mode = TDIFC_MODE_NULL;
    int next_mode = TDIFC_MODE_NULL;

    int tileX = 0;
    int tileY = 0;
    int grid_size = 0;
    int block_size = 0;

    static float *newtile = NULL;
    float *tile = NULL;

    float *d_in = 0;
    float *d_out = 0;

    /* Perform the appropriate computation based on the current
       mode. */
    mode = tdifc_ec_get_mode(ec);

    if (mode == TDIFC_LIB_INVERT_M_IDX_INIT) {
        tileX = *((int *)tdifc_ec_get_param(ec,
                TDIFC_LIB_INVERT_P_IDX_TILEX));
        tileY = *((int *)tdifc_ec_get_param(ec,
                TDIFC_LIB_INVERT_P_IDX_TILEY));
        newtile = (float*)malloc(sizeof(float) * tileX * tileY);
        next_mode = TDIFC_LIB_INVERT_M_IDX_INVERT;
    } else if (mode == TDIFC_LIB_INVERT_M_IDX_INVERT) {
        /* Get parameter values and inputs. */
        tileX = *((int *)tdifc_ec_get_param(ec,
                TDIFC_LIB_INVERT_P_IDX_TILEX));
        tileY = *((int *)tdifc_ec_get_param(ec,
                TDIFC_LIB_INVERT_P_IDX_TILEY));
        grid_size = *((int *)tdifc_ec_get_param(ec,
                TDIFC_LIB_INVERT_P_IDX_GRID_SIZE));
        block_size = *((int *)tdifc_ec_get_param(ec,
                TDIFC_LIB_INVERT_P_IDX_BLOCK_SIZE));
        tdifc_tc_read(tc, TDIFC_LIB_INVERT_I_IDX_INPUT1, &tile);
        cutilSafeCall(cudaMalloc((void**)&d_out, sizeof(float) *
                tileX * tileY));
        cutilSafeCall(cudaMalloc((void**)&d_in, sizeof(float) *
                tileX * tileY));

        if (0 == newtile || 0 == d_in || 0 == d_out) {
            printf("Could not allocate memory: host = %p,
                    device = %p\n", newtile, d_in);
            return;
        }

        cudaMemset(d_out, 0, sizeof(float) * tileX * tileY);
        cudaMemcpy(d_in, tile,  sizeof(float) * tileX * tileY,
                cudaMemcpyHostToDevice);
        printf("%s\n", cudaGetErrorString(cudaGetLastError()));

        {
            dim3 grid;
            dim3 block;
            grid.x = grid_size;
            grid.y = grid_size;
            block.x = block_size;
            block.y = block_size;

            stencil<<<grid, block>>>( d_in, d_out );
            cudaMemcpy(newtile, d_out,  sizeof(float) * tileX *
                    tileY, cudaMemcpyDeviceToHost);
        }

        /* Write the output. */
        tdifc_tc_write(tc, TDIFC_LIB_INVERT_O_IDX_OUTPUT1,
                &newtile);
        next_mode = TDIFC_LIB_INVERT_M_IDX_INVERT;
    } else {
        tdifcuda_lib_invert_invoke_error("Invalid mode");
    }

    /* Set the next mode. */
    tdifc_ec_set_mode(ec, next_mode);
}
```

3.3.3. Developing Schedulers

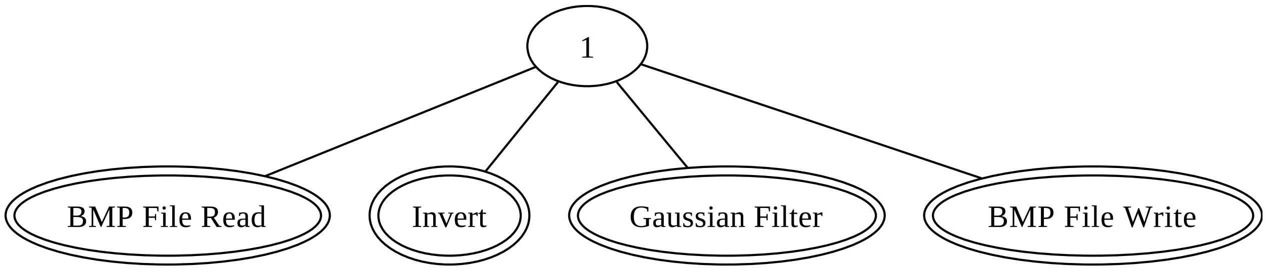


Figure 6: GST Representation of Canonical Schedule for the Application Graph in Figure 2

A *scheduling transformation* transforms an application dataflow graph into a representation that contains a sequence of actor firings and associated control logic that can be used to execute the dataflow graph. In CFDF graphs, the `enable` and `invoke` methods effectively allow executing an actor in a given mode only if sufficient input data is available. Testing for such data sufficiency is performed through run-time checks implemented using the `enable` method. A *guarded execution* of a CFDF actor *A* is a single invocation of *A* that is conditional upon the `enable` method first returning `true`. If *A* is *not* enabled for execution at a given point in time, then a guarded execution of *A* at that time can be viewed as a `NOP` (no operation).

A simple scheduling transformation for CFDF, called the *canonical scheduler*, is one that generates a guarded execution of every actor in the CFDF graph, and sequences these guarded executions in some arbitrary order. The resulting schedule, called a *canonical schedule*, can then be repeated until the entire input data set is processed, a required number of outputs is generated or some other stopping criterion is met.

We represent a canonical schedule for the application shown in **Figure 2** using a *generalized schedule tree* (*GST*), as shown in **Figure 6**. GSTs provide a dataflow-model-independent representation of schedules, which can be utilized as an input to subsequent stages of a design flow, such as simulation and code synthesis [7]. An internal node of a GST denotes a loop count (the number of times to execute the associated subtree), while a leaf node points to an actor. The execution of a schedule involves traversing the GST in a depth-first manner, and during this traversal, the sub-schedule rooted at any internal node is executed as many times as specified by the loop count of that node. In the GST in **Figure 6**, double peripheries around leaf nodes indicate guarded execution of the corresponding actors.

The following code demonstrates how a canonical schedule can be implemented.

```
/* Canonical schedule repeated for iter number of times */
for (i = 0; i < iter; i++) {
    /* Guarded execution of BMP file read */
    if (tdifc_ec_enable_check(tdifcuda_lib_bmp_file_read_ec,
            tdifcuda_lib_bmp_file_read_tc)) {
        tdifc_ec_invoke(tdifcuda_lib_bmp_file_read_ec,
                tdifcuda_lib_bmp_file_read_tc);
    }
    /* Guarded execution of invert */
    if (tdifc_ec_enable_check(tdifcuda_lib_invert_ec,
            tdifcuda_lib_invert_tc)) {
        tdifc_ec_invoke(tdifcuda_lib_invert_ec,
                tdifcuda_lib_invert_tc);
    }

    /* Guarded execution of gfilter */
    if (tdifc_ec_enable_check(tdifcuda_lib_gfilter_ec,
            tdifcuda_lib_gfilter_tc)) {
        tdifc_ec_invoke(tdifcuda_lib_gfilter_ec,
                tdifcuda_lib_gfilter_tc);
    }

    /* Guarded execution of BMP file write */
    if (tdifc_ec_enable_check(tdifcuda_lib_bmp_file_write_ec,
            tdifcuda_lib_bmp_file_write_tc)) {
        tdifc_ec_invoke(tdifcuda_lib_bmp_file_write_ec,
                tdifcuda_lib_bmp_file_write_tc);
    }
}
```

In the TDIF environment, the designer has the flexibility to integrate, apply, and reuse more sophisticated schedulers in the processes of design space exploration and implementation.

3.4. Instrumentation Techniques

Performance measurement of GPU-accelerated code must in general take into account overall application performance, including the contributions due to any associated GPPs (general purpose processors), and other types of processing resources in the target platform.

Our approach to performance measurement and instrumentation distinguishes between intra-actor and inter-actor code performance, as well as performance of actor code as it executes on different types of resources in a heterogeneous implementation platform. Orthogonal to the optimization of CUDA actors on a GPU-enabled platform, scheduling determines the resource on which each actor executes and the order of execution among actors that share the same processing resource. Thus, scheduling typically has a significant impact on software synthesis quality. Given a library of component modules (e.g., dataflow actors), and a formal application specification, software synthesis selects a subset of modules and configures the interactions among them to implement a given application.

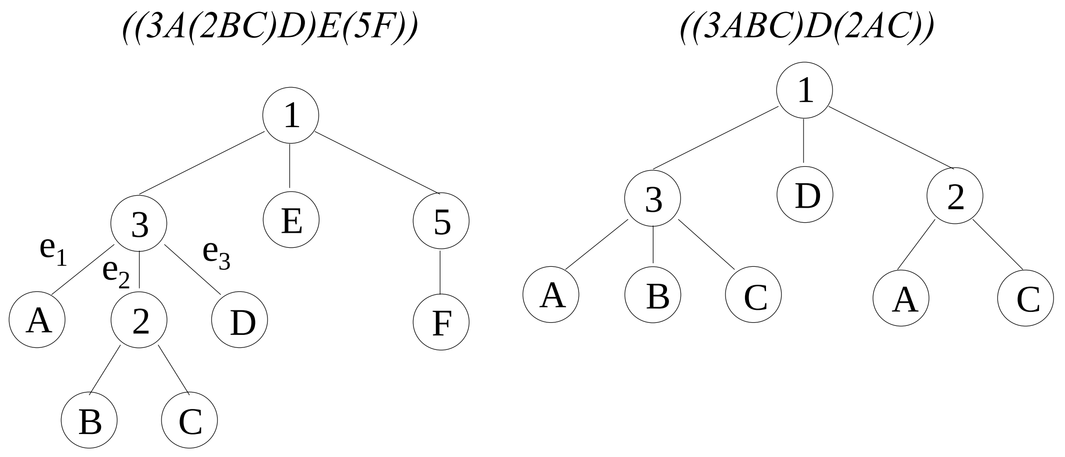


Figure 7: GST Representation for Schedules

In addition to performance, memory usage is often highly sensitive to scheduling decisions due to the data-driven property of dataflow graph execution. We have developed instrumentation methods to assess trade-offs between performance and memory usage and between intra-actor and inter-actor code performance in implementations that are synthesized from DIF and TDIF specifications. By applying these methods, designers can tune library module implementations as well as strategies for scheduling and buffer management based on characteristics and constraints of the given application and platform. Such tuning can be performed efficiently in our new DIF/TDIF framework given the formal dataflow graph structure that is enforced by the framework.

3.4.1. Instrumented Schedule Trees

Our approach to instrumentation in TDIF is designed to support the following key requirements: (a) no change in functionality (instrumentation directives should not change application functionality); (b) operations for adding and removing instrumentation points should be performed by designers in a way that is external to actors (i.e., does not interfere with or require modification of actor code); and (c) instrumentation operations should be *modular* so that they can be mixed, matched, and migrated with ease and flexibility. Such a structured, dataflow-integrated approach to instrumentation provides significant benefits compared to the ad-hoc approaches to instrumentation that are typically used in multimedia system implementation.

Instrumentation support in TDIF builds on the *generalized schedule tree* (GST) representation, which provides a standard graphical format for representing a broad class of dataflow graph schedules [7]. In a GST, each leaf node refers to an actor invocation, and each internal node n represents an expression that is interpreted as an iteration count I_n for the associated sub-tree (that is, execution of the sub-tree rooted at n is repeated I_n times).

((3A(2BC)D)E(5F))

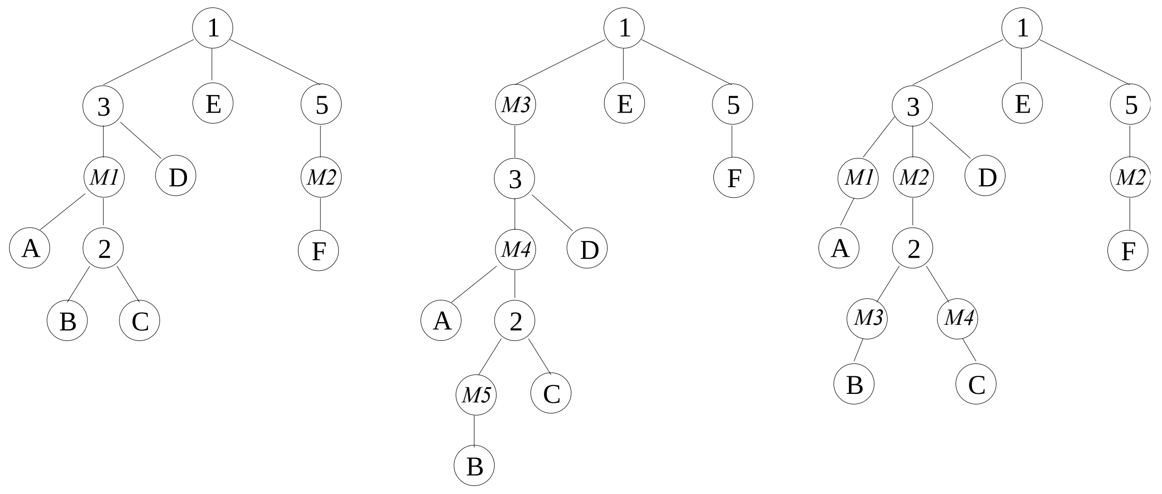


Figure 8: Examples of IGSTs

In its *schedule tuning mode*, TDIF allows designers to augment the GST representation with functional modules, encapsulated as *instrumentation nodes* (*INs*), which are dedicated to instrumentation tasks. Like iteration nodes, instrumentation nodes are incorporated as internal nodes. We refer to GSTs that are augmented with instrumentation nodes as *instrumented GSTs* (*IGSTs*). The instrumentation tasks associated with an instrumentation node are in general applied to the corresponding IGST sub-tree. **Figure 7** shows an example of two GSTs, and **Figure 8** shows the example of IGSTs for the schedule *((3A(2BC)D)E(5F))*. In **Figure 8**, *M1… M5* represent instrumentation nodes.

An IGST allows software synthesis for a schedule together with instrumentation functionality that is integrated in a precise and flexible format throughout the schedule. Upon execution, software that is synthesized from an IGST produces profiling data (e.g., related to memory usage, performance or power consumption) along with the output data that is generated by the source application.

An instrumentation node in general has two associated functions, *pre* and *post*, which represent instrumentation-related computations (e.g., system calls, accesses to specialized memory locations, counter accesses, etc.) that are to be carried out just before and after, respectively, the associated IGST sub-tree executes.

Depending on the desired instrumentation functionality, one or both of the functions *pre* and *post* can be used. If both are used (e.g., for performance measurement), such an instrumentation node can be viewed as providing *interval instrumentation*, whereas if only one is used (e.g., to record memory usage), it can be viewed as *point instrumentation*.

Instrumentation nodes therefore provide a formal, dataflow-integrated approach for specifying instrumentation functionality in a manner that flexibly interacts with but is cleanly separated from the code (schedule and actor code) that it interacts with. Such orthogonalization across scheduling, actor, and instrumentation functionality is a key strength of TDIF, which adds to the modularity and productivity features offered by the environment.

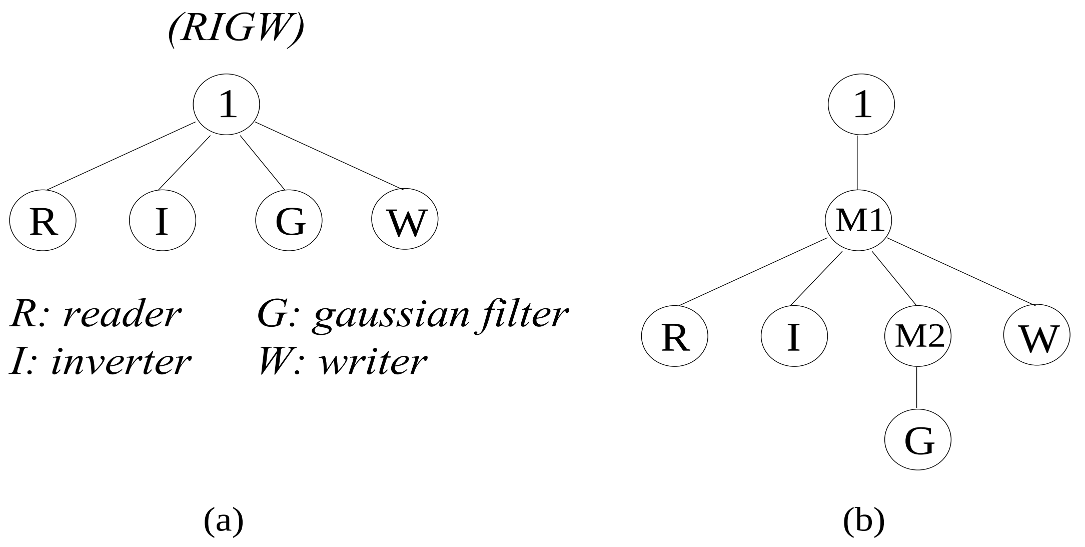


Figure 9: (a) GST of Canonical Schedule for the Gaussian Filtering Application (b) The corresponding IGST

3.5. DIFML

We have developed a design for the DIFML format which is an XML-based format for exchanging information between DIF and other tools and languages, and more generally, between arbitrary pairs of dataflow environments. Associated software plug-ins for DIF have also been implemented. There are different elements in the DIFML package, and these elements are listed hierarchically when formulating DIFML descriptions. The element at the highest level is the `graph`, while `topology` and `interface` are lower level elements. Under `topology`, there are three elements at the same level: `node`, `edge`, and `interface`.

For each element, there are three kinds of attributes: `implicitAttributes`, `builtInAttributes` and `userDefinedAttributes`. ImplicitAttributes are those attributes necessary and inherent to the element, such as the id of a node. BuiltInAttributes are attributes that are recognized as part of the DIF language, typically through corresponding reserved words or other kinds of language constructs.

3.5.1. XML format

The extensible markup language, widely known as XML, is a markup language that was created by the World Wide Web Consortium (W3C) to overcome limitations of HyperText Markup Language (HTML). Like HTML, XML is based on SGML — the Standard Generalized Markup Language. Although SGML has been used in the publishing industry for decades, its perceived complexity intimidated many people that otherwise might have used it. XML was designed with the Web in mind.

A major advantage of XML is that one can encode document information more precisely compared to HTML. This means that programs processing these documents can "understand" them much better and therefore process the information in ways that are not possible for ordinary text processors.

One major application of XML is to make web pages with decent layout that are universally accessible, regardless of browser type. XML also lets one check whether or not optional features are present, and allows for invocation of alternative code to take care of cases where such features are missing.

XML is a promising candidate for carrying data associated with high level text based languages for subsequent use. XML itself is designed to be self-descriptive, which ensures that all of the information from the original file can be understood by other applications. XML tags are not predefined by users. It can be convenient for users to design appropriate tags to describe the context of the information being exchanged.

Representing different languages using a common XML format allows for integrated use of heterogeneous languages within a design flow, thereby allowing designers to combine the unique strengths and features associated with different languages.

Interfacing between the DIF framework and other languages and tools can be achieved using DIFML, which is an XML-based format associated with DIF.

3.5.2. The DIFML format

As described previously, the dataflow interchange format (DIF) is proposed as a standard approach for specifying and integrating arbitrary dataflow-based semantics for DSP system design [2], and The DIF language (TDL) is an accompanying textual design language for high-level specification of signal-processing-oriented dataflow graphs.

In order to describe DIFML, we introduce a number of concepts associated with the general XML format: node, element, attribute and tags. A node is a part of the hierarchical structure that makes up an XML document. "Node" is a generic term that applies to any type of XML document object, including elements, attributes, comments, processing instructions, and plain text. A tag is a markup construct that begins with < and ends with >. Tags come in three flavors: start-tags, for example `<section>`, end-tags, for example `</section>`, and empty-element tags, for example `<line-break/>`. An element is a logical component of a document. An element either begins with a start-tag and ends with a matching end-tag, or consists only of an empty-element tag. The characters between the start- and end-tags, if any, are the element's content, and may contain markup, including other elements, which are called "child elements". An attribute is a markup construct consisting of a name/value pair that exists within a start-tag or empty-element tag.

DIFML is designed as an XML-based format for exchanging information between TDL and other tools and languages, and more generally, between arbitrary pairs of dataflow environments. There are different elements in DIFML and these elements are listed in a hierarchical way. The element at the highest level is `graph`, while `topology` and `interface` are lower level elements. Under `topology`, there are three elements at the same level: `nodes`, `edges` and `interface`. For each element, there are three kinds of attributes: `implicitAttributes`, `builtInAttributes` and `userDefinedAttributes`. ImplicitAttributes are those attributes necessary and inherent to the element, such as the id of a node. BuiltInAttributes are attributes that are recognized as part of the DIF language, typically through corresponding reserved words or other kinds of language constructs. For example, for an edge element in an SDF model within a DIF graph (i.e., within a graph that is defined with the `sdf` keyword), there are three kinds of builtInAttributes: the production rate, consumption rate, and delay. UserDefinedAttributes are attributes that users add to selected elements at their own discretion. The following is a simple example of an SDF model in the DIFML format. For conciseness, we just show part of the associated DIFML file.

```
<?xml version=' 1. 0 ' encoding='UTF−8 ' ?>
<difml xmlns=' http: //www. ece .umd . edu/DIFML '>
        <graph>
                <implicit Attributes>
                        <name val=' dat2cd ' />
                        <type val='SDFGraph ' />
                </ implicit Attributes >
        <topology>
                <nodes>
                        <node>
                                < implicit Attributes >
                                        <id val='A' />
                                </ implicit Attributes >
                                <builtInAttributes>
                                        <nodeWeight type='DIFNodeWeight ' />
                                </ builtInAttributes >
                                <userDefinedAttributes>
                                        <attribute name=' output ' type='Edge ' val=' e1 ' />
                                        < attribute name=' readerFP ' type='DIFParameter ' val=' reade r ' />
                                </ userDefinedAttributes >
                        </node>
                </nodes>
                <edges>
                        <edge>
                                < implicit Attributes >
                                        <id val=' e1 ' />
                                        <sourceId val='A' />
                                        <sinkId val='B' />
                                </ implicit Attributes>
                                <builtInAttributes>
                                        <edgeWeight comsumption=' [ 2 ] ' delay=' 0 '
                                                        production=' [ 1 ] ' type='SDFEdgeWeight ' />
                                </ builtInAttributes>
                        </ edge>
                </ edges>
        </ topology>
        <interface>
                <port>
                        < implicit Attributes>
                                <direction id=' InA ' nodeId='A' val=' IN ' />
                        </ implicit Attributes>
                </ port>
                <port>
                        < implicit Attributes>
                                <direction id='OutE ' nodeId='E ' val='OUT' />
                        </ implicit Attributes>
                </ por t>
        </ interface>
        </graph>
<!−−Automatically generated from DIF file−−>
</ difml>
```

As shown in the above example, each DIFML element contains an opening tag, a closing tag, and some content. The opening tag begins with a left angle bracket (<), followed by an element name that contains letters and numbers (but no spaces), and finishes with a right angle bracket (>). Following the content is the closing tag, which exhibits the same spelling and capitalization as the opening tag, but with one small change: a / appears right before the element name. Note that there is an element named `node`. This name is in correspondence with the related definition in the DIF language, and has different meaning with from the "node" concept in XML terminology, which is a generic concept that applies to any type of XML document object.

Currently, the DIFML parser supports several major dataflow models that are recognized in the DIF language, including SDF [3], cyclo-static dataflow (CSDF) [8], core functional dataflow (CFDF) [4], parameterized synchronous dataflow (PSDF) [9], CAL dataflow (CALDF) [10], and multidimensional synchronous dataflow (MDSDF) [11].

3.5.3. The DIFML Package

The DIFML package is developed using Java and can be used for converting file formats between DIF and DIFML. That is, given a DIF file based on a specific dataflow model, such as SDF or CFDF, the DIFML package can transform it into the corresponding DIFML format and store the output into a DIFML file (i.e., `*.difml`). On the other hand, given a DIFML file, the DIFML package can transform it into the corresponding DIF format and store such format into a DIF file (i.e., `*.dif`). The bridge between the DIF file and the DIFML file is the DIF intermediate representation.

In the project deliverables, two commands are provided to transform between the DIF and DIFML formats. To transform from the DIF format to the DIFML format, we have introduced `df2dfml`:

`df2dfml` <input dif file>

The output will be stored in the file `<file>.difml`, where `<file>.dif` is the name of the original DIF file.

Similarly, to transform from the DIFML format to the DIF format, we have introduced `dfml2df`:

`dfml2df` <input difml file>

The output will be stored in the file `<file>.dif`, where `<file>.difml` is the name of the original DIFML file.

4. Results and Discussion

The actor performance implemented in CUDA is tuned according to the profiling results generated by CUDA Visual Profiler as well as theoretical analysis of the application. Some types of schedules represented as GSTs can be generated, traversed and documented automatically by the DIF package. Using such information, we specify where and what the instrumentation points are in the GST, and insert those points into the corresponding software synthesis result manually. Then, performance is measured after compiling the program and running the program with relevant input data sets.

To demonstrate how the TDIF scheduling-based instrumentation framework handles instrumentation, we use five different filter sizes to configure the `Gfilter` actor and implement the Gaussian filtering application, as shown in **Figure 2**, in C and CUDA. In addition to performance measurement, we conduct experiments on memory management efficiency as well as on trade-offs between performance and inter-actor context switch overhead. We will show later in this section that the schedule can lead to greatly improved performance if it is well-tuned — e.g., by allowing actors to fully utilize resources without unnecessary waiting.

We change the filter sizes for the `Gfilter` actor and denote them as 5X5, 11X11, 21X21, 25X25 and 37X37. The schedule shown in **Figure 9** (a) contains references to the `BMP_File_Read` and `BMP_File_Write` actors that are implemented in C, and `Invert` and `Gfilter` actors that are implemented in CUDA. As illustrated in **Figure 9** (b), two schedule nodes, *M1* and *M2*, are used for *point instrumentation* and *interval instrumentation*, respectively.

The IN *M1* measures three types of bandwidth — Host to Device Bandwidth, Device to Host Bandwidth and Device to Device Bandwidth — before the execution of the application. The IN *M2* monitors the execution time of the `Gfilter` actor.

```
Device 0: GeForce GTX 260
Host to Device Bandwidth, 1 Device(s), Paged memory
Transfer Size (Bytes) Bandwidth(MB/s)
33554432 1234.5

Device to Host Bandwidth, 1 Device(s), Paged memory
Transfer Size (Bytes) Bandwidth(MB/s)
33554432 960.7

Device to Device Bandwidth, 1 Device(s)
Transfer Size (Bytes) Bandwidth(MB/s)
33554432 97519.1
```

Figure 10: Results from Quick Mode Profiling

Table 1: Performance Comparison for the `Gfilter` Actor Implemented in C and CUDA

Filter size	5X5	11X11	21X21	25X25	37X37
CUDA (ms)	4.228	4.874	10.257	12.759	21.72
C (ms)	50	280	1080	1540	3310
Speed up	11.83	57.45	105.29	120.70	152.39

Table 2: Performance Comparison for the Gaussian Filtering Application Implemented in C and CUDA

Filter size	5X5	11X11	21X21	25X25	37X37
CUDA (ms)	70	80	140	115	130
C (ms)	70	295	1100	1550	3340
Speed up	1	3.69	7.86	13.48	25.69

As a key building block for constructing libraries of instrumentation nodes, *point instrumentation* is a function with arguments for different measurement purposes. This function supports three modes, which are relevant for bandwidth testing, as well as for testing of other performance characteristics.

- Quick mode: performs a quick measurement.
- Range mode: measures a user-specified range of values.
- Shmoo mode: performs an intense shmoo of a large range of values.

To experiment in quick mode, *M1* is specified to run a bandwidth test in quick mode, and the corresponding function call is inserted into the IGST during software synthesis. **Figure 10** shows the measurement result for quick mode at the very beginning of execution. Even though the underlying instrumentation function is restricted to run its bandwidth test in one of three modes (during a given function call), the definition of *M1* is flexible so that the function can be called multiple times with different arguments if needed. The cooperation between *M1* and the corresponding IN implementation makes instrumentation convenient, flexible, and efficient.

Three main aspects affects the performance:

- The implementation of `Gfilter`;
- the memory usage (buffer requirement) for dataflow graph edges; and
- the schedule.

Given five different filter sizes for the `Gfilter` actor, our experiments involving performance instrumentation include measurements of application performance and the performance on only the `Gfilter` actor with GPU acceleration. Here, *M1* is specified as an interval instrumentation operation. This interval instrumentation operation measures overall application performance, and *M2* measures the performance of the `Gfilter` actor. As shown in **Table 1**, the CUDA implementations exhibit superior performance compared to the corresponding C implementations in these experiments.

Table 2 provides a performance comparison for the overall Gaussian filtering application that is implemented in C and CUDA. The application-level speedups, while still significant, are consistently less than the corresponding actor-level speedups. This is due to factors such as context switch overhead and communication cost for memory movement, which are associated with overall schedule coordination in the application implementations.

5. Conclusion

In this project, we have developed and delivered novel software tools for design and implementation of layered sensing and signal processing systems. The *Targeted DIF* (TDIF) environment is a GPU-targeted software synthesis tool, which is based on the dataflow interchange format (DIF) framework, and provides a unique integration of dynamic dataflow modeling; retargetable actor construction; software synthesis; and instrumentation-based schedule evaluation and tuning. The DIFML package is a software package for the DIFML format, which is an XML-based format for exchanging information between DIF and other tools and languages, and more generally, between arbitrary pairs of dataflow environments. We have also presented and delivered application case studies to demonstrate the utility of the TDIF and DIFML environments.

6. References

[1] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*. Springer, 2010.

[2] C. Hsu, M. Ko, and S. S. Bhattacharyya, “Software synthesis from the dataflow interchange format,” in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, Dallas, Texas, September 2005, pp. 37–49.

[3] E. A. Lee and D. G. Messerschmitt, “Synchronous dataflow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[4] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, “Functional DIF for rapid prototyping,” in *Proceedings of the International Symposium on Rapid System Prototyping*, Monterey, California, June 2008, pp. 17–23.

[5] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf

[6] C. Donnelly and R. Stallman, *Bison – The Yacc-compatible Parser Generator*, August 2010.

[7] M. Ko, C. Zissulescu, S. Puthenpurayil, S. S. Bhattacharyya, B. Kienhuis, and E. Deprettere, “Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation,” *IEEE Transactions on Signal Processing*, vol. 55, no. 6, pp. 3126–3138, June 2007.

[8] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, “Cyclo-static dataflow,” *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, February 1996.

[9] B. Bhattacharya and S. S. Bhattacharyya, “Parameterized dataflow modeling for DSP systems,” *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, October 2001.

[10] J. Eker and J. W. Janneck, “CAL language report, language version 1.0 — document edition 1,” Electronics Research Laboratory, University of California at Berkeley, Tech. Rep. UCB/ERL M03/48, December 2003.

[11] E. A. Lee, “Multidimensional streams rooted in dataflow,” in *Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, Florida, January 1993, pp. 20–22.

LIST OF ACRONYMS

ADT	Abstract Data Type
API	Application Programming Interface
BMP	Bitmap Image File
CALDF	CAL Dataflow
CFDF	Core Functional Dataflow
Cg	C for Graphics
CSDF	Cyclo-static Dataflow
CUDA	Compute Unified Device Architecture
DIF	Dataflow Interchange Format
DIFML	Dataflow Interchange Format Markup Language
DSP	Digital Signal Processing
EC	Execution Context
FIFO	First-in-first-out
GPP	General Purpose Processor
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HTML	HyperText Markup Language
MDSDF	Multidimensional Synchronous Dataflow
NOP	No Operation

NVCC	NVIDIA CUDA Compiler
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
PSDF	Parameterized Synchronous Dataflow
SDF	Synchronous Dataflow
SDK	Software Development Kit
SGML	the Standard Generalized Markup Language
TC	Topological Context
TDIF	Targeted DIF
TDIFSyn	TDIF Synthesis
TDL	The DIF Language
TDP	The DIF Package
XML	Extensible Markup Language